Functions in MATLAB

MATLAB has a number of predefined functions and you can write any function yourself. You can do it either straight from the command line, or you can write a function M-file. The latter is sometimes necessary for certain tasks we will learn later, and often more convenient if you need to use the same function repeatedly: you just call the function script.

For polynomials and rational functions we can generate a function value for any single input number (remember: function is a rule that when given a number produces another number) using the basic operations on numbers (arithmetic) you learned before. For example, if I want to evaluate a function \( y = x^2 + 5x + 7 \) for \( x = 2.75 \), it would be

```matlab
>> x=2.75;
>> y= x^2 + 5*x + 7
y =
 2.8313e+001
```

or better:

```matlab
>> y = x*x + 5*x + 7
y =
 2.8313e+001
```

for a rational function, such as \( y = \frac{x+1}{2x^2-x-2} \):

```matlab
>> y=(x+1)/(2*x*x - x - 2)
y =
 3.6145e-001
```

Note that \( x \) has to have a value! Computers do not do symbols, only numbers. Unless you have a value in \( x \) it does not know what to calculate and gives you an error:

```matlab
>> clear x
>> y=(x+1)/(2*x*x - x - 2)
??? Undefined function or variable 'x'.
```

This takes care of all your polynomials and rational functions, for exponentials, logarithms, trigonometric and their inverses, and hyperbolic functions MATLAB has the following commands (built-in functions), some of which are listed in the following table:
## Built-in Functions

<table>
<thead>
<tr>
<th>function</th>
<th>MATLAB</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>x</td>
<td>\sqrt{x}$</td>
</tr>
<tr>
<td>$\sqrt[n]{x}$</td>
<td>$\text{nthroot}(x)$</td>
<td>do not use $x^{(1/n)}$</td>
</tr>
<tr>
<td>$e^x$</td>
<td>$\exp(x)$</td>
<td></td>
</tr>
<tr>
<td>$\ln(x)$</td>
<td>$\log(x)$</td>
<td>natural log (base $e$)</td>
</tr>
<tr>
<td>$\log(x)$</td>
<td>$\log_{10}(x)$</td>
<td>common log (base 10)</td>
</tr>
<tr>
<td>$\log_2(x)$</td>
<td>$\log_2(x)$</td>
<td></td>
</tr>
<tr>
<td>$\sin(x)$</td>
<td>$\sin(x)$</td>
<td>input in radians</td>
</tr>
<tr>
<td>$\sin(x)$</td>
<td>$\sin(x)$</td>
<td>input in degrees</td>
</tr>
<tr>
<td>$\cos(x)$</td>
<td>$\cos(x)$</td>
<td>input in radians</td>
</tr>
<tr>
<td>$\cos(x)$</td>
<td>$\cosd(x)$</td>
<td>input in degrees</td>
</tr>
<tr>
<td>$\tan(x)$</td>
<td>$\tan(x)$</td>
<td>input in radians</td>
</tr>
<tr>
<td>$\tan(x)$</td>
<td>$\tand(x)$</td>
<td>input in degrees</td>
</tr>
<tr>
<td>$\cot(x)$</td>
<td>$\cot(x)$</td>
<td>input in radians</td>
</tr>
<tr>
<td>$\cot(x)$</td>
<td>$\cotd(x)$</td>
<td>input in degrees</td>
</tr>
<tr>
<td>$\arcsin(x)$</td>
<td>$\text{asin}(x)$</td>
<td>output in radians</td>
</tr>
<tr>
<td>$\arcsin(x)$</td>
<td>$\text{asin}(x)$</td>
<td>output in degrees</td>
</tr>
<tr>
<td>$\arccos(x)$</td>
<td>$\text{acos}(x)$</td>
<td>output in radians</td>
</tr>
<tr>
<td>$\arccos(x)$</td>
<td>$\text{acosd}(x)$</td>
<td>output in degrees</td>
</tr>
<tr>
<td>$\arctan(x)$</td>
<td>$\text{atan}(x)$</td>
<td>output in radians</td>
</tr>
<tr>
<td>$\arctan(x)$</td>
<td>$\text{atand}(x)$</td>
<td>output in degrees</td>
</tr>
<tr>
<td>$\arccot(x)$</td>
<td>$\text{acot}(x)$</td>
<td>output in radians</td>
</tr>
<tr>
<td>$\arccot(x)$</td>
<td>$\text{acotd}(x)$</td>
<td>output in degrees</td>
</tr>
<tr>
<td>$\sinh(x)$</td>
<td>$\sinh(x)$</td>
<td></td>
</tr>
<tr>
<td>$\cosh(x)$</td>
<td>$\cosh(x)$</td>
<td></td>
</tr>
<tr>
<td>$\tanh(x)$</td>
<td>$\tanh(x)$</td>
<td></td>
</tr>
<tr>
<td>$\coth(x)$</td>
<td>$\coth(x)$</td>
<td></td>
</tr>
<tr>
<td>$\arcsinh(x)$</td>
<td>$\text{asinh}(x)$</td>
<td></td>
</tr>
<tr>
<td>$\arccosh(x)$</td>
<td>$\text{acosh}(x)$</td>
<td></td>
</tr>
<tr>
<td>$\arctanh(x)$</td>
<td>$\text{atanh}(x)$</td>
<td></td>
</tr>
<tr>
<td>$\text{acoth}(x)$</td>
<td>$\text{acoth}(x)$</td>
<td></td>
</tr>
</tbody>
</table>

### Examples:

```matlab
>> x=pi/4;
>> sin(x)
ans =
    0.70711e-001
>> x=45;
>> sind(x)
ans =
    0.70711e-001
>> asind(ans)
ans =
    4.5000e+001
```
**Arrays and “Vectors”**

Because functions are used to express dependence of one quantity on another, we often need them over a range of points, rather than just a single point \( x \). For example, we are used to plotting a function \( y = f(x) \) as a continuous line for some interval of \( x \) values. However, remember that a plot of a function is really pairs of \((x, y)\) values. That is also how we have to think about functions on the computer.

On a computer we cannot specify a continuous interval. We can only give it a set of values \( x \), which are spaced by some finite step. For such a set of \( x \) we can evaluate the corresponding set of values \( y \). The function is then represented as two sets of corresponding \( x \) and \( y \) numbers.

With what we have done so far, we would have to repeatedly calculate the function values for different values of \( x \) (e.g. \( x_1, x_2, \ldots, x_N \)) and store them in different variables \((y_1, y_2, \ldots, y_N)\). There is much more efficient way to do this, using *arrays* of numbers.

An array is really just a string of numbers. Individual numbers in an array are called *elements* of an array and they are distinguished by their *index*. For example I can have an array \( x \) with 10 elements in it. The first element would be \( x(1) \), the second \( x(2) \) and the last \( x(10) \). That way you can have only one variable (e.g. \( x \)) but a whole lot of numbers in it, which can be called by their indeces.

In general, arrays can have more than one dimension. One dimensional (1-D) array is essentially equivalent to a vector. Two-dimensional (2-D) array, where each element would have two indices, is equivalent to a matrix. We will discuss vectors and matrices in much greater detail later. For our purposes now we only need to be able to define one dimensional arrays of numbers for the purpose of calculating, plotting and later on doing some other numerical manipulations with functions.

**Defining an array**

There are several ways to define an array:

1. **Element by element.**

Write the elements in square brackets separated by a space:

```matlab
>> x=[1 2 3 4 5 6 7 8 9 10]
```

\[
x =
\begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10
\end{bmatrix}
\]

```matlab
>> x=[1.2 2.5 3.2 4.4 5 6.6 7e-2 9.1 12.38478 3.1415 2.789 1 0 -2]
```

\[
x =
\begin{bmatrix}
1.2000e+000 & 2.5000e+000 & 3.2000e+000 & 4.4000e+000 & 5.0000e+000 & 6.6000e+000 \\
7.0000e-002 & 9.1000e+000 & 1.2385e+001 & 3.1415e+000 & 2.7890e+000 & 1.0000e+000 \\
0 & -2.0000e+000
\end{bmatrix}
\]
From the second example you can see that you can mix integers, decimals, scientific notation at will, but MATLAB will convert all the numbers to decimal (and scientific in this case, because I have set format short e).

Also note that when MATLAB cannot fit your array all on the same line, it will write out what “Columns” the numbers are at. This is where the vector character comes in. The way we defined our array is equivalent to defining a row vector. Row vector has one row and multiple columns.

We could have also defined our array as a column vector by separating the numbers by semicolon instead of space:

```matlab
>> x=[1;2;3;4;5;6;7;8;9;10]
```

```
x =
1
2
3
4
5
6
7
8
9
10
```

and we have our array ordered in a column. Whether you have row or column vector does not matter now (it will be important when we talk about matrices etc.) and it is up to you which one you prefer. I personally prefer columns.

### 2. Using a colon “:” operator.

Writing A:B means from A to B by 1, i.e. it is the same as [A A+1 ... B].

If we want to space our elements by a different step than one, we use two colon operators:

A:D:B will produce an array starting at A all the way to B with step D. (Sometimes a whole number of D steps will not take you exactly to B, in that case the array will end at the closest smaller number to B)

```matlab
>> x=1:10
```

```
x =
1  2  3  4  5  6  7  8  9 10
```
\[ x = 1:2:10 \]

This will work not only for integers, but for any numbers:

\[ x = 1.28e-3:1e-4:4.5e-3 \]

Now if we want to have the array in a column, instead of a row, we have to transpose it. That is done using an “’” (apostrophe or quote) operator on the vector:

\[ x = x' \]
3. Using linspace() function

The linspace (for “linear space”) function is useful for generating arrays with specified number of points. It is a computer function (not mathematical function), but like a mathematical function has parentheses and takes arguments, in this case two or three:

- `linspace(A, B)` generates a row vector of 100 linearly equally spaced points between A and B.
- `linspace(A, B, N)` generates N points between A and B. (For N < 2, linspace returns B).

For example to generate 10 points between 1 and 5:

```matlab
>> x=linspace(1,5,10)
x =
Columns 1 through 6
1.0000e+000 1.4444e+000 1.8889e+000 2.3333e+000 2.7778e+000 3.2222e+000
Columns 7 through 10
3.6667e+000 4.1111e+000 4.5556e+000 5.0000e+000
```

And if you prefer a column, you can again transpose it:

```matlab
>> x=x'
x =
1.0000e+000
1.4444e+000
1.8889e+000
2.3333e+000
2.7778e+000
3.2222e+000
3.6667e+000
4.1111e+000
4.5556e+000
5.0000e+000
```

Indexing Arrays

To display a particular element of an array, type and array name with the index of that particular element in parentheses. For example, the 5th element of the above `x` array is

```matlab
>> x(5)
an =
2.7778e+000
```

The first element would be

```matlab
>> x(1)
an =
1
```

If I want the last element I actually do not need to know its number (which equals the length of the array), but use ‘end’ as an index:

```matlab
>> x(end)
an =
5
```
However, if I want to know the length, there is a `length()` function, which takes the array name as an argument:

```
>> length(x)
ans =
  10
```

and tells me in this case that array \( x \) has 10 elements.

**Array Operations and Functions**

Having defined a set of \( x \) values, we would like to use them to represent mathematical functions. That means to evaluate a function value for each of the \( x \) array elements to get a corresponding array of \( y \) function values.

The basic array operations or *element-by-element* operations, such as multiplication, division and raising to a power, are done using slightly different commands that those we used previously for numbers. The reason is that the same operations must be defined for matrices and they work somewhat differently. MATLAB uses the operators * (multiplication), / or \( \backslash \) (division) and \(^\) as *matrix operators*, while *array operators* for element-by-element operations are preceded by a dot ‘.’. The matrix operators work on numbers, because each number is in fact a 1x1 matrix. Addition (+) and subtraction (−) are the same, as are multiplication (*) and division (/) of the whole array by a constant (number).

Another important condition is that when performing an operation that involves more than one array, all the arrays have to have the same number of element (this makes sense, how else would you do element by element operation), but they also have to be either all rows or all columns. The result then will be an array with the same number of elements that is again either row or column.

Here is the list of *array operations* and corresponding symbols:

<table>
<thead>
<tr>
<th>operation</th>
<th>operator</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+</td>
<td>a+5, a+b</td>
</tr>
<tr>
<td>subtraction</td>
<td>−</td>
<td>a-2.7, a-b</td>
</tr>
<tr>
<td>multiplication by a constant</td>
<td>*</td>
<td>1.25*a</td>
</tr>
<tr>
<td>multiplication of two arrays</td>
<td>element-by-element</td>
<td>.* a.*a (same as a.^2), a.*b</td>
</tr>
<tr>
<td>division by a constant</td>
<td>/</td>
<td>a/8.3145, b/3</td>
</tr>
<tr>
<td>division of two arrays</td>
<td>element by element</td>
<td>./ a./b, 1./a</td>
</tr>
<tr>
<td>power</td>
<td>element by element</td>
<td>.^ a.^4 (same as a.*a.*a.*a), b.^2.5</td>
</tr>
<tr>
<td>exponential (except ( e^a ))</td>
<td>element by element</td>
<td>.^ 2.^a, 10.^b</td>
</tr>
</tbody>
</table>

If this is confusing, forget *, / and ^ and always use .* , ./ and .^ (the array operators) instead. It works, even for the simple arithmetic with plain numbers (they also happen to be special cases on one element arrays).

When we get to matrices, we will see when to use the matrix operators.
Examples:

```matlab
>> x=[0 1 2 3]
x =
    0 1 2 3
>> x+2
ans =
    2 3 4 5
>> x+x
ans =
    0 2 4 6
>> x+2*x
ans =
    0 3 6 9
>> x*x
??? Error using ==> mtimes
Inner matrix dimensions must agree.
>> x.*x
ans =
    0 1 4 9
>> x^2
??? Error using ==> mpower
Inputs must be a scalar and a square matrix.
>> x.^2
ans =
    0 1 4 9
>> x/x
ans =
    1
```

Note: the matrix division actually works (does not give you an error). But it also does not give you an array, just a number!

```matlab
>> x./(1+x)
ans =
    0 5.0000e-001 6.6667e-001 7.5000e-001
```

The built-in mathematical functions listed in the “Built-in Functions” table above work for arrays just as the array operations in element-by-element fashion. For example, \( \sin(x) \) will calculate the \( \sin \) for each element of the array.

```matlab
>> sin(x)
an =
    0 8.4147e-001 9.0930e-001 1.4112e-001
```
Now we can use these array operations and built-in functions to represent any mathematical function for a any range of independent variables of your choice. For example the function

\[ y = 2 \sin(x^2) + 3 \quad \text{for} \quad 0 \leq x \leq 1 \]

might look like this:

```matlab
>> x=0:0.2:1;
>> y=2*sin(x.^2)+3;
>> x
x =
    0    0.2000    0.4000    0.6000    0.8000    1.0000
>> y
y =
    3.0000    3.0800    3.3186    3.7045    4.1944    4.6829
```

You can see that we have pairs of \((x, y)\), in other words for each \(x\) we have a \(y\), which is exactly how we think about functions in the “finite resolution” world (i.e. on the computer). It is important to realize that the function values \(y\) are now also an array. Therefore, we can pull out the individual ones by their indices:

```matlab
>> y(1)
an =
    3
>> y(4)
an =
    3.7045e+000
>> y(end)
an =
    4.6829e+000
```

This is important, because all the manipulations with functions later on will be based on the fact that they are represented by arrays of numbers.

**Plotting Functions**

Plotting functions is one of the most important and useful things we can do with them. There is no substitute for seeing how they look like. MATLAB can generate an amazing variety of different plots and graphs.

The most basic way to graph a function is to use the `plot()` function, which has different forms, depending on the input arguments. If \(y\) is an array, `plot(y)` produces a piecewise linear graph of the elements of \(y\) versus the index of the elements of \(y\).

```matlab
>> plot(y)
>>
```

This will open a new window, called “Figure 1” and give you a plot from 1 to 5 on the \(x\)-axis (because you have 5 array elements, with indices 1, 2, 3, 4 and 5) and the corresponding values on the \(y\)-axis.
If you specify two arrays as arguments, `plot` produces a graph of $x$ versus $y$. For example, to plot a sine of $x$ values ranging from 0 to $2\pi$:

```
>> x=0:pi/10:2*pi;
>> y=sin(x);
>> plot(x,y)
```

The plot command will also take expressions as long as they produce an array of the right size. We could do

```
>>plot(x,sin(x));
```

You can beautify your plot, by labeling the axes and adding a title:

```
>> xlabel('x')
>> ylabel('Sine of x')
>> title('Plot of the Sine Function')
```
**Specifying Line Styles and Colors**

By default, the plot command will produce a blue line. Note however, that the line only connects your \((x, y)\) data points. It is possible to specify color, line styles, and markers (such as plus signs or circles) when you plot your data using the plot command:

\[
\text{plot}(x,y,\text{\textquote{color_style_marker}})
\]

`color_style_marker` is a string containing from one to four characters (enclosed in single quotation marks) constructed from a color, a line style, and a marker type. The strings are composed of combinations of the following elements:

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>'c'</td>
<td>cyan</td>
</tr>
<tr>
<td></td>
<td>'m'</td>
<td>magenta</td>
</tr>
<tr>
<td></td>
<td>'y'</td>
<td>yellow</td>
</tr>
<tr>
<td></td>
<td>'r'</td>
<td>red</td>
</tr>
<tr>
<td></td>
<td>'g'</td>
<td>green</td>
</tr>
<tr>
<td></td>
<td>'b'</td>
<td>blue</td>
</tr>
<tr>
<td></td>
<td>'w'</td>
<td>white</td>
</tr>
<tr>
<td></td>
<td>'k'</td>
<td>black</td>
</tr>
<tr>
<td>Line style</td>
<td>' - '</td>
<td>solid</td>
</tr>
<tr>
<td></td>
<td>' - - '</td>
<td>dashed</td>
</tr>
<tr>
<td></td>
<td>':'</td>
<td>dotted</td>
</tr>
<tr>
<td></td>
<td>' - - '</td>
<td>dash-dot</td>
</tr>
<tr>
<td></td>
<td>no character</td>
<td>no line</td>
</tr>
<tr>
<td>Marker type</td>
<td>'+'</td>
<td>plus mark</td>
</tr>
<tr>
<td></td>
<td>'o'</td>
<td>unfilled circle</td>
</tr>
<tr>
<td></td>
<td>'*'</td>
<td>asterisk</td>
</tr>
<tr>
<td></td>
<td>'x'</td>
<td>letter x</td>
</tr>
<tr>
<td></td>
<td>'s'</td>
<td>filled square</td>
</tr>
<tr>
<td></td>
<td>'d'</td>
<td>filled diamond</td>
</tr>
<tr>
<td></td>
<td>'v'</td>
<td>filled upward triangle</td>
</tr>
<tr>
<td></td>
<td>'v'</td>
<td>filled downward triangle</td>
</tr>
<tr>
<td></td>
<td>'&lt;'</td>
<td>filled right-pointing triangle</td>
</tr>
<tr>
<td></td>
<td>'&gt;'</td>
<td>filled left-pointing triangle</td>
</tr>
<tr>
<td></td>
<td>'p'</td>
<td>filled pentagram</td>
</tr>
<tr>
<td></td>
<td>'h'</td>
<td>filled hexagram</td>
</tr>
<tr>
<td></td>
<td>no character or none</td>
<td>no marker</td>
</tr>
</tbody>
</table>

If you specify a marker type but not a line style, only the marker is drawn. For example,

\[
\text{plot}(x,y,\text{\textquote{ks}})
\]

plots black squares at each data point, but does not connect the markers with a line.
On the other hand, the statement

\[ \text{plot}(x,y,'r:+') \]

plots a red-dotted line and places plus sign markers at each data point:
**Adding Plots to an Existing Graph**

The MATLAB hold command enables you to add plots to an existing graph. When you type

```
hold on
```

now MATLAB does not replace the existing graph when you issue another plotting command; it adds the new data to the current graph, rescaling the axes if necessary. For example the following will put a Cosine function over the previous sine plot

```
>> plot(x,y,'ks')
>> plot(x,y,'r:+')
>> hold on
>> plot(x,cos(x),'g-^')
```

The `legend` command provides an easy way to distinguish individual plots:

```
>> legend('Sine','Cosine')
```

![Figure 1](image-url)

The `hold on` can be reversed by `hold off` command. After that any new plot command will again erase all the previous plots.
**Figure Windows**

Graphing functions automatically open a new figure window if there are no figure windows already on the screen. If a figure window exists, it is used for graphics output. If there are multiple figure windows open, the one that is designated the “current figure” (the last figure used or clicked in) is used.

To make an existing figure window the current figure, you can click the mouse while the pointer is in that window or you can type

```
figure(n)
```

where \( n \) is the number in the figure title bar. The results of subsequent graphics commands are displayed in this window. If the figure number \( n \) does not exist, this command will open a new figure window, with a number \( n \).

If you just want to open a new figure window and make it the current figure, simply type

```
figure
```

this figure window will have the number one higher that the highest number figure window already open.

**Clearing the Figure for a New Plot**

When a figure already exists, most plotting commands clear the axes and use this figure to create the new plot. However, these commands do not reset figure properties, such as the background color or the colormap. If you have set any figure properties in the previous plot, you might want to use the `clf` command with the reset option:

```
clf reset
```

before creating your new plot to restore the figure’s properties to their defaults.

If is a good practice to preemptively clear the figures in your scripts, in particular if you overlay multiple plots. Having statements such as

```
figure(1);hold off; clf reset;
```

before you start plotting will ensure that all the old stuff will always be erased if you run your scripts multiple times (which happens a lot when you are writing and testing them).
**Writing Your Own Functions**

You can write a function the same way you write a script. In fact the function is also an M-file (with the extension .m) Function M-file, however, has a different syntax than the script M-file. It always looks like this:

```
function [ output_args ] = Untitled( input_args )

%UNTITLED Summary of this function goes here
% Detailed explanation goes here

end
```

In fact, when you start writing a function M-file, you go to the top menu and select File → New → Function MATLAB will already have this written for you. You just sort of fill in the blanks. Let’s try it on several examples.

As the first and simple example, let us write a function for \( f(x) = e^{-\lfloor x \rfloor} [1 + \cos(2x) + \sin(2x)] \).

There is only one input argument and one output argument. We will call the output “\( y \)” the input “\( x \)” and give the function name “my_function”. This is how the function script would look like:

```
function y = my_function(x)

% my_function calculates the following function:
% y = exp(-abs(x)) * (1 + cos(2*x) + sin(2*x))

y = exp(-abs(x)) .* (1 + cos(2*x) + sin(2*x));

% remember array element-by-element operations
% to make this work on an array of numbers x
% you need to use './.' when multiplying tow arrays

end
```

Now you have to save this file. It is important that the filename (with .m after it) is the same as the name of the function, in this case my_function.m

If you go to File → Save As MATLAB will already offer you this filename as default:
So all you have to do is click **Save**.

Now we can try it out. First we need to define the input array, call the function and assign the output to a variable. Then we will plot the function.

To show you that the input and output variables can have any name (i.e. different than \(x\) and \(y\) that we used for the definition of the function), we will call the input \(z\) and the output \(f\).

```matlab
>> z=-5:0.1:5;
>> f=my_function(z);
>> plot(z,f,'r');
>>
```

And this is what we get:
Function can have more than one input (and also output) arguments. For example, we can make a more general function by using parameters, such as \( f(x) = e^{-dx} [1 + b \cos(2x) + c \sin(2x)] \), which is a somewhat different function for different values of \( a \), \( b \) and \( c \), but it is necessary to write only one function file for all.

\[
\begin{align*}
\text{function } & y = my\text{-function2}(x,a,b,c) \\
& %\text{calculates } \ y = \exp(-a^2\text{abs}(x)) \times (1+b\cos(2x)+c\sin(2x)) \\
& y = \exp(-a^2\text{abs}(x)) \times (1+b\cos(2x)+c\sin(2x)) \\
& \text{end}
\end{align*}
\]

Save this as my_function2.m and let’s try it out. Since we are only interested in plotting this function for different sets of parameters, we can take a few shortcuts:

```matlab
>> z=-5:0.1:5; >> a=0.1;b=2;c=0; >> plot(z,my_function2(z,a,b,c)) here I am calling my_function2 directly from the inside of plot function >> a=1;b=0;c=-1; >> hold on >> plot(z,my_function2(z,a,b,c),'r') >> a=-0.2;b=2;c=2; >> plot(z,my_function2(z,a,b,c),'g') >> legend('a=0.1,b=2,c=0','a=1,b=0,c=-1','a=-0.2,b=2,c=2') >>
```

![Plot of the function with different parameters](image)
More Gymnastics with Arrays

1. Cutting out a part of an array
This can be done using a colon operator. Remember it means range. Therefore
A(i1:i2) will display array elements A(i1), A(i1+1), ... ,A(i2-1), A(i2)

```
>> A=1:10
A =
   1     2     3     4     5     6     7     8     9    10
>> A(3:6)
ans =
   3     4     5     6
```

You can assign the part of an array to another variable, and get another array that you can work with, calculate functions for etc.

```
>> B=A(5:end)
B =
   5     6     7     8     9    10
```

2. Joining arrays
For row arrays, to join two together, you just write them back to back in square brackets and get a row array

```
>> A=[1:2;3]
A =
   1
   2
   3
>> B=[4;5;6]
B =
   4
   5
   6
>> C=[A;B]
C =
   1
   2
   3
   4
   5
   6
```

If you have columns, you have to separate them by a semicolon to get a column array:

```
>> A=[1;2;3]
A =
   1
   2
   3
>> B=[4;5;6]
B =
   4
   5
   6
>> C=[A;B]
C =
   1
   2
   3
   4
   5
   6
```
3. Deleting an element or elements

The easiest thing to go that is to assign an empty array “[ ]” (empty square brackets) to the elements you want to delete.

```
>> C
C =
1
2
3
4
5
6
>> C(end) = []
C =
1
2
3
4
5
```

This deleted the last element \( C(\text{end}) \) and your array is now one shorter. In the same way you can delete element or elements in the middle. If you have more than one element, use colon to give a range:

```
>> C(2:4) = []
C =
1
5
```

This works exactly the same for rows and columns.

4. Logical Indexing and the \texttt{find()} Function

This is useful for finding elements or parts of your arrays that are greater, equal or less than some value. If you test an array, say for whether it is greater than zero, the answer will be a logical array, with zeros corresponding to the elements where it is not true, and ones where it is true:

```
>> X = -5:5
X =
-5  -4  -3  -2  -1  0   1   2   3   4   5
>> I = X > 0
I =
0   0   0   0   0   0   1   1   1   1   1
```

You can use the logical array to index the part of your array where the logical expression is true. A logical array index designates the elements of an array \( X \) based on their \textit{position} in the indexing array, \( I \), not their value. That means inserting \( I \) into \( X \) will give me only the part of \( X \) which corresponds to ones in \( I \) (or the elements of \( X > 0 \)):

```
>> X(I)
ans =
1   2   3   4   5
or I could have done instead
```

```
>> X(X > 0)
ans =
1   2   3   4   5
```
The logical indexing can be used together with the find function, whose general syntax is

\[ I = \text{find}(\text{logical\_array}, K) \]

and which returns the first \( K \) indices corresponding to the non-zero values of the logical array. \( K \) must be a positive integer. The logical array can be the result of a logical expression where you can test for the desired property of your array.

For example, I want to find the first element of the following array

\[ X = -5:5 \]

\[ X = \begin{array}{ccccccccc}
-5 & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 & 5
\end{array} \]

which is greater than zero

\[ \text{find}(X>0,1) \]

\[ \text{ans} = 7 \]

and it tells me that it is the seventh element. Is that true?

\[ X(7) \]

\[ \text{ans} = 1 \]

Looks right.

Now if I want all the elements that are greater than zero, I drop the number \( K \) and do simply

\[ I = \text{find}(X>0) \]

\[ I = \begin{array}{cccccc}
7 & 8 & 9 & 10 & 11
\end{array} \]

and again we can check whether it is true. The variable “\( I \)” is now an array of indices, so I can just call the elements of the array \( X \) corresponding to the indices in \( I \):

\[ X(I) \]

\[ \text{ans} = \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5
\end{array} \]

and we have the whole positive part of the array.

What these tricks allow you to do is to write all kinds of discontinuous and crazy functions for different intervals of \( x \). For example, an important function is the so-called Heaviside function, which is one for \( x \geq 0 \), and zero otherwise:

\[ y = \begin{cases} 
1 & x \geq 0 \\
0 & x < 0
\end{cases} \]

This function obviously has a discontinuity at 0. To write a function M-file for this, one way is to take advantage of the logical indexing
and to try it out:

```
>> x=-10:0.1:10;
>> plot(x,Heavi(x),'or')
>> xlabel('x');ylabel('y');title('Heaviside function');
```